

Final Report: Implementation of the Bluetooth stack for software defined radio, with a view to sniffing and injecting packets

Dominic Spill

May 14, 2007

Supervisor: Andrea Bittau

This report is submitted as part requirement for the MSci degree in Computer Science at University College London. It is substantially the result of my own work except where explicitly indicated in the text.

The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

This project sets out to implement some of the functionality of the Bluetooth protocol using the GNU Radio framework and a Universal Software Radio Peripheral (USRP) to replace dedicated Bluetooth hardware. The functionality is specifically the ability to read packets from a connection between two device, although in this case the devices should be unaware that the packets are being read by a third party.

The ability to receive and demodulate a stream of data from a Bluetooth connection is achieved, using the USRP to receive the signal, and the GNU Radio Gaussian Minimum Shift Keying (GMSK) demodulator to convert the complex signal to binary. The control and settings of these parts of the system are found through experimentation with signal processing.

The binary data is then converted to packets and data is extracted from them in order to gather information about the devices in the piconet so that the MAC address and value of the clock of the piconet can be obtained. Data from higher levels of the protocol stack can also be extracted from these packets, which could lead to possible attacks on a system.

The ability to sniff packets gives an insight into the protocol at a lower level than has been investigated by previous academic work. This moves attacks, that were theoretical in nature, much closer to the goal of a practical attack.

Contents

1	Introduction	1
1.1	Background	1
1.1.1	Bluetooth	1
1.1.2	GNU Radio	2
1.1.3	The Universal Software Radio Peripheral	3
1.1.4	Why use GNU Radio for Bluetooth?	3
1.2	Aims and outcomes	4
1.2.1	Goals	4
1.2.2	Results	4
2	Background	5
2.1	Definitions	5
2.2	Properties of Bluetooth devices and connections	6
2.2.1	FEC1/3	7
2.2.2	FEC2/3	7
2.3	Packet format	7
2.3.1	Access code	7
2.3.2	Packet header	8
2.3.3	Packet types	9
2.3.4	Payload	11
2.3.5	Data whitening	12
2.4	Bluetooth connection protocol	12
2.5	Passive vs active attacks	13
2.6	Related work	14

3	Finding the signal	15
3.1	Why does the signal need to be found?	15
3.2	Details of the signal	15
3.3	Experiments	16
3.3.1	Random stabbing in the dark	17
3.3.2	Test packets	17
4	Interpreting the signal	20
4.1	Deciphering test packets	20
4.2	Real packets	21
4.3	32 possible UAPs	23
5	Implemented functions	27
5.1	Experimental setup	27
5.1.1	Physical layout	27
5.1.2	Data flow of the signal	28
5.2	GNU Radio gr-bluetooth block	28
5.2.1	Modules	29
5.3	How the modules work	30
5.3.1	Access Code	31
5.3.2	Header	31
5.3.3	Payload	32
5.4	GNU Radio scripts	32
5.5	bccmd	33
5.6	Testing	34
5.6.1	Access code	34
5.6.2	Header	34
5.6.3	Payload	35
6	Evaluation	36
6.1	The USRP's role in Bluetooth hacking	36
6.2	Exploiting the Bluetooth protocol	37
6.2.1	Finding the MAC	37
6.2.2	Finding the clock	38

6.2.3	Putting it all together	38
7	Conclusions	40
7.1	Have the aims been met?	40
7.2	Why is this not a full implementation?	40
7.2.1	Finding the signal	40
7.2.2	Lack of frequency hopping	41
7.3	Software written	41
7.4	Where does this leave Bluetooth, GNU Radio and the USRP?	42
8	Further work	43
8.1	Gathering more data from sniffed packets	43
8.2	Hopping	44
8.3	Using commodity hardware	44
8.4	Mounting an attack	44
A	System manual	46
A.1	Command line tools	46
A.1.1	bccmd	46
A.1.2	packet_finder	46
A.1.3	packet_sniffer	47
A.2	GNU Radio Bluetooth (gr-bluetooth)	47
A.2.1	sniffer	47
A.2.2	bluetooth_LAP_sniffer.py	47
A.2.3	bluetooth_sniffer.py	47
A.2.4	file_sniffer.py	48
B	User manual	49
B.1	Command line tools	49
B.1.1	bccmd	49
B.1.2	packet_finder	50
B.1.3	packet_sniffer	50
B.2	GNU Radio Bluetooth (gr-bluetooth)	51
B.2.1	sniffer	51

B.2.2	bluetooth_sniffer.py	51
B.2.3	file_sniffer.py	51
B.3	Frequency / gain	51
B.3.1	Frequency	51
B.3.2	Gain	52
C	Test results	53
C.1	Packet codes and error checks	53
C.1.1	acgen()	53
C.1.2	hecgen()	53
C.1.3	crcgen()	53
C.2	Packet sniffing and data extraction	54
C.2.1	sniff_ac()	54
C.2.2	UAP_from_hec()	54
C.2.3	unwhiten()	54
D	Interim report	55
D.1	Progress made to date	55
D.2	Further work	57
E	Code	58
E.1	Access code	58
E.1.1	codeword()	58
E.1.2	acgen()	60
E.1.3	checkac()	63
E.1.4	sniffac()	64
E.2	Header	66
E.2.1	hecgen()	66
E.2.2	UAP from HEC	67
E.2.3	Unwhiten header	68
E.3	Payload	69
E.3.1	crcgen()	69
E.4	Common functions	70
E.4.1	reverse	70

E.4.2	convert to one LSB per byte format	70
-------	--	----

Chapter 1

Introduction

This work sets out to implement some of the functions of the Bluetooth protocol using the hardware designed for the GNU Radio project, which gives complete control over the radio to the software controlling it. A single frequency is monitored for packets, which are captured, demodulated and interpreted. The process gives valuable insight into the way that Bluetooth works at a lower level than has been studied before.

Research has concentrated on attacks at a higher level of the protocol, such as encryption, key exchange, brute forcing the pin or extracting data from a previously paired device[4, 5, 14], but many of these assume that packets can be sniffed from an arbitrary connection, this work takes steps toward achieving this goal and enabling a practical demonstration of these theoretical attacks.

1.1 Background

1.1.1 Bluetooth

The Bluetooth communication protocol was developed by the “Bluetooth Special Interest Group (SIG) ... comprised of ... Agere, Ericsson, IBM, Intel, Microsoft, Motorola, Nokia and Toshiba” [1] in 1998. It was designed as a cable replacement protocol and is widely used for low bitrate data transfer

such as voice and small data communications. It is mostly marketed as a mobile adhoc technology, and is therefore present in most modern mobile telephone handsets, PDAs and notebook computers.

There are alternatives to Bluetooth, such as 802.11 and zigbee, which are both IEEE standards and operate in the same 2.4GHz band. They are used for high(54Mbps) and low(250kbps)[3] data transfer rates respectively. 802.11 is a very common technology which is often used for LAN connectivity as a replacement for ethernet. There are many known exploits and attacks against it which allow it to be snooped or used as an up-link to the internet without authorisation. Zigbee is less prolific and is aimed at the field of sensornets and home automation[2], its uses will likely grow in the coming years, but as yet it is relatively unheard of.

Bluetooth can be implemented on a single chip for a cost of around \$3[6]. This means that it is often cheaper to add to a device than 802.11, zigbee or even cabled communication. This all adds up to make it an extremely common technology, especially in the workplace, which makes it interesting from a security point of view.

1.1.2 GNU Radio

The GNU Radio project is a framework for performing digital signal processing and controlling software defined radio. It allows for rapid development of signal processing application by joining “blocks” together to form a pipeline, taking the signal from reception to useful output.

A “block” is a small step of processing within a pipeline, they range from file sources and sinks, to demodulators and from packet parsers to PAL television decoders. Most blocks are written in C++ with a frontend in python, the scripts that link the pipeline stages are written in python. It is a future goal of the project to allow entire applications to be written in C++, this will reduce the overheads and allow some of the slower pipelines to run in

real time.

1.1.3 The Universal Software Radio Peripheral

The Universal Software Radio Peripheral is the hardware designed to be used with GNU Radio software, it was designed and built by Ettus research [17]. The device consists of a mainboard with digital signal processors on it, along with connected daughterboards which are specific to various frequency bands. The RFX-2400 is aimed at the 2.4GHz to 2.5GHz industrial, scientific and medical band. This is the frequency band that Bluetooth uses, and is therefore the only daughterboard needed. It comes with two connections, one for transmission and receptions, and one that only receives, for the majority of uses only transmission or reception will be used at any one time, so the combined port shall be used.

1.1.4 Why use GNU Radio for Bluetooth?

Bluetooth devices are small, cheap and easy to get hold of, so the decision to implement the protocol using a complex piece of rare and expensive hardware may seem an odd one. The decision stems from the problem of trying to sniff packets of other communications because the commodity hardware ignores packets which are not destined for that device. The debug mode of the Cambridge Silicon Radio (CSR) devices will allow for the MAC of the device to be altered, but this would only allow communications to be snooped if the clock signal and MAC of the system to be sniffed are known. This is unlikely to be known without internal knowledge of the system concerned, which would probably provide easier avenues of attack. This work is concerned with sniffing packets from an arbitrary communication, which was established at some point before sniffing was started.

The GNU Radio is useful for this because it allows the frequency of the signal to be set at any point and is not locked into hopping or a specific protocol. This means that the hopping pattern of a device can be followed

without having to pair devices, giving the option of an entirely passive attack. The framework, coupled with the USRP device, allows packets to be captured and studied, giving an insight into the Bluetooth protocol at the radio and baseband layers that has not been possible for the average user until now.

1.2 Aims and outcomes

1.2.1 Goals

The aim of this work is to produce software with the ability to sniff packets from an arbitrary Bluetooth connection, giving the chance to attack a communication and extract user data from the connection. This software should use the USRP device, along with the GNU Radio framework, to allow information about the connection, as well as the devices taking part, to be read from the packets that are received.

1.2.2 Results

The software written is able to receive a signal on a single frequency and demodulate this into a stream of binary data. The binary stream is then split into packets by searching for known patterns within the data. Then information can be extracted, from these packets, about the devices involved in the communication. This brings the ability to attack communications between Bluetooth devices a lot closer, and shows that there are only a small number of steps to take before a practical attack can be perpetrated.

Chapter 2

Background

The Bluetooth specification is a large and complicated document, and a considerable amount of time needs to be taken in order to understand the way by which connection and communication takes place. Version 1.2 of the document is 1200 pages in length, and covers all aspects of the protocol. Key parts of the specification, which relate to the work later in this document, are described below.

2.1 Definitions

Several phrases are used here, and throughout this document, to describe functions, devices and variables, these are defined here for ease of reference.

USRP Universal software radio peripheral, see Section 1.1.3

RFX-2400 The 2.4GHz daughterboard to the USRP, locked to the Industrial, Scientific and Medical frequency band

Access Code An identifier at the start of the packet

HEC Header Error Check

CRC Cyclic Redundancy Check

UAP Upper Address Part

LAP Lower Address Part

NAP Non-significant Address Part

LT_ADDR Logical Transport Address, a unique address given to each device within a piconet

Clock Clock refers to the internal clock of a Bluetooth device

Piconet A personal area network formed by one master and many slave devices

FEC Forward Error Correction, applied to packets to mitigate transmission errors

LFSR Linear Feedback Shift Register, used to calculate HEC, CRC and whitening

GFSK Gaussian Frequency Shift Keying, the modulation system used by the Bluetooth radio layer

GMSK Gaussian Minimum Shift Keying, a modulation technique that can be tuned to function as GFSK

2.2 Properties of Bluetooth devices and connections

Each Bluetooth enabled device has a MAC address assigned by the IEEE from the pool of available six byte IDs, this is split into three parts, LAP for the least significant three bytes, UAP for the next byte, and NAP for the two most significant devices. The device also maintains a clock which is a 28 bit counter that increments once every 312.5us (a 3.2KHz signal).

All communications are packet based and packets are sent at a rate of 1600 per second. Data is modulated using GFSK modulation, where a frequency shift of +115KHz represents a binary 1, and -115KHz represents a binary 0. This data is often protected with forward error correction which comes in two forms, FEC1/3 and FEC2/3.

2.2.1 FEC1/3

FEC 1/3 outputs three times the number of input bits. It does this by replicating each input bit three times, so an input stream of “01010” becomes “000111000111000”. When decoding the packet any bit errors can be avoided by simply averaging the bits received in groups of three.

2.2.2 FEC2/3

FEC 2/3 encoding outputs a five bit Hamming code for every ten data bits that are taken as input. This can be used to check for bit errors, and correct many small errors. If the input is not divisible by ten then the last part is padded with ‘0’ bits to bring it up to length. The algorithm for producing the Hamming code is defined by a LFSR in vol 2, part B, section 7.5[7].

2.3 Packet format

Packets are made up of three sections, access code, header and payload. All packets have an access code to inform other devices which piconet the packet is meant for, and most have a header and payload with data in them. The header defines the type of packet that is being transmitted, and the data stored in the payload is formatted accordingly.

2.3.1 Access code

Each packet has an access code which lets all listening devices know which piconet it is being transmitted to. The access code is normally 72 bits long and is generated from the three least significant bytes of the MAC address (LAP) of the master device. The ID packet, used for finding devices within range, is just a 68 bit access code as it needs to be transmitted in a shorter amount of time. The access code is formed by creating a BCH(64, 30) code, or “syncword”, from the LAP and a six bit Barker sequence set by the most significant bit of the LAP. A preamble and a trailer are added based on the

first and last bits of the syncword respectively. This is shown in Figure 2.1.

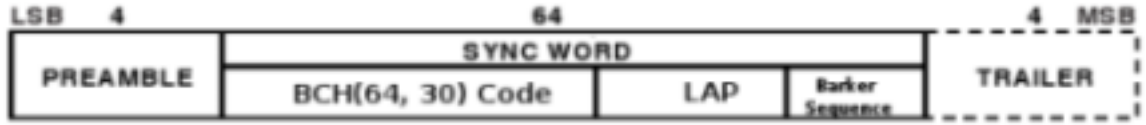


Figure 2.1: Composition of an access code [7]

The preamble is set to 1010 or 0101 if the LSB of the codeword is 1 or 0 respectively. The Barker code and trailer are set together as 1100101010 or 0011010101 if the MSB of the LAP is 1 or 0 respectively.

When sniffing packets the LAP can be taken straight from the access code, and then checked against the BCH(64, 30) code to confirm that it is a packet and not just noise. The start of a packet itself can be found by looking for the preamble and Barker code + trailer sequences of bits at the correct distance apart. The access code is always the same for a given piconet, it does not change with the device that is transmitting.

2.3.2 Packet header

Every packet, other than the ID packet, has a packet header directly following the access code which gives the device ID within the piconet, the type of the packet and flow, ack and sequence information. This is followed by a header error check (HEC) which is calculated from the ten bits of header data by a linear feedback shift register (LFSR), which is initialised with the UAP of the master device. The packet header is encoded with FEC1/3 and also whitened with data based on the state of the clock at time of transmission, it is then encoded with forward error correction (FEC). Figure 2.2 shows the layout of the data within the packet header before whitening or FEC is applied.

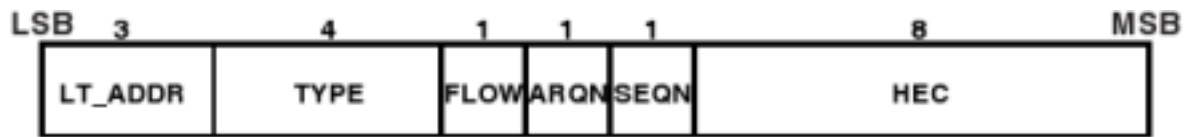


Figure 2.2: Composition of a packet header [7]

2.3.3 Packet types

There are 18 different packet types, which fall into four groups based on the type of data transfer that occurs. SCO and eSCO connections often, but not always, carry voice data, while ACL links carry file data and control information for the higher layers of the protocol. The specific details of these packet are given here for reference only, the details are not required for a full understanding of the work described later in the document.

common packets

Common packets that are used for control and connection establishment are:

ID - a 68bit packet used to signal the start of connection establishment.

FHS - a 240bit packet used to transfer MAC and clock data between devices in order for them to synchronise.

POLL - used to check that a device is still available and listening for data, can be used as a low level ping.

NULL - used as a response to acknowledge packet transmission.

Asynchronous Connection-Less (ACL) packets

ACL packets contain a variable length payload followed by a CRC, they are used for data transfer which is not timing critical. There are two types, each with a number of different length variants.

DH1 - a packet containing up to 27 bytes of data, with a CRC at the end. DH packets do not have forward error correction applied.

DH3 - identical to DH1 packet, but takes three times as long to transmit and carries up to 183 bytes of data.

DH5 - identical to DH1 packet, but takes five times as long to transmit and carries up to 339 bytes of data.

DM1 - a packet carrying up to 17 bytes of data, with a CRC at the end. The data is encoded with FEC2/3.

DM3 - the same as DM1, but takes three times as long to transmit, carries up to 121 bytes of data.

DM5 - as DM1, but takes five times as long to transmit, carries up to 224 bytes.

AUX1 - This is defined as an axillary packet, and is reserved for future use.

Synchronous Connection Oriented (SCO) packets

SCO packets are used to transfer voice or data at 64kb/s, there are four packets, the choice of which is based on length of data to be sent, as most of them have fixed length data fields, with the DV packet being used for transferring voice and data in one packet with a variable field length. All packets take the same amount of transmission time.

HV1 - carries 10 bytes of data protected by FEC1/3 encoding.

HV2 - is a variant of the HV1 packet but uses FEC2/3, and can contain up to 20 bytes of data.

HV3 - is not protected by FEC encoding and therefore can carry 30 bytes of user data.

DV - carries 10 bytes of voice data followed by a variable length data field. The data field is protected by FEC2/3 encoding.

Extended Synchronous Connection Oriented (eSCO) packets

The eSCO packets are used for the same purposes as SCO packets, but contain a CRC at the end of the data section. There are three types based on

length, but no equivalent to the DV packet type. The length of the data in the packets is decided at the establishment of the connection.

EV3 - carries up to 30 bytes of data which is not protected by FEC encoding.

EV4 - carries up to 120 bytes of data, protected by FEC2/3 encoding.

EV5 - carries up to 180 bytes of data which is not protected by FEC encoding.

2.3.4 Payload

The payload is different for each packet type, some, such as the DH1/2/3 or DM1/2/3 packets, have a payload header followed by a variable length data field and a CRC. Others, such as the EV3/4/5 packets, have a fixed length of data which is chosen at the establishment of communication, followed by a CRC. The DV packet is a combination with an 80 bit voice field followed by a variable length data field and CRC. For voice data there are HV1/2/3 packets which have a fixed length and no CRC, they are also never retransmitted.

For the packets that contain a variable length data field there is a payload header which also contains a link ID; this is only used for asynchronous packets, i.e. packets that are retransmitted in the event of failure.

A number of the packet types have forward error correction applied to the payload, some use the FEC1/3 scheme where each bit is simply repeated three times, an average can be taken in order to get the original data back. Other packets use the FEC2/3 scheme which produces five bits of parity data for every ten bits of the payload; the packet is divided into blocks of ten bits followed by the parity data for that block, the final block is padded with '0's to produce a block of ten bits.

2.3.5 Data whitening

The packet header and payload are whitened on every packet, except the ID packet which has neither header nor payload, by XORing the bits with a data from a LFSR. This LFSR takes six bits of the clock value as input and produces a stream of binary to XOR with the packet data. Due to the input being limited the LFSR only has 64 possible starting values, and 127 possible output values, the register cannot be set or reach a state where it is 0. This is only done once a connection has been established and both devices are using a common clock.

2.4 Bluetooth connection protocol

Bluetooth devices have a pattern of frequencies that they transmit on set by their MAC address and clock signal. If two devices wish to communicate they must be synchronised in their hopping and, as Bluetooth is a one-to-many protocol, they must be able to address the packets for the intended recipient. This is done through the inquiry process which takes place at the start of communications to establish hopping patterns, identifiers and encryption keys.

One of the devices initiates the communication, this is designated as the master device, the other device is the slave; this can be changed later in communications for devices that only operate as a slave such as wireless headsets. In order to communicate with a device it must first be found, this is done by hopping through only 32 frequencies at a higher rate, 3200 times per second, transmitting only the ID packet, (see section 2.3.1).

When a device receives the ID packet on a frequency that it is listening to it replies on the same frequency 625us later. The master device will return to the frequency in order to listen for the reply. The slave device responds with an FHS packet which contains all of the MAC address and clock data, this can be used to calculate the hopping pattern and sniff packets from the

device, it is therefore ignored by many devices to avoid denial of service attacks and snooping.

If the devices wish to communicate after the inquiry phase is complete then the master is able to copy the hopping pattern of the slave device, using the data in the FHS packet, and begin to establish a connection.

2.5 Passive vs active attacks

At some point while devising an attack against a system the decision must be made as to how to perform the attack. The possibilities generally fall into two camps, passive and active attacks.

Passive - A passive attack is one in which the victims are not interfered with, and can therefore be completely unaware of the attack. This style of attack also does not require data to be transmitted, which reduces the complexity of it. However services provided by Bluetooth devices cannot be hijacked without communicating with them, so a passive attack has limited use. For intercepting voice and file transfers that use Bluetooth a passive attack will suffice.

Pros Undetectable, could potentially listen to multiple simultaneous communications.

Cons May be computationally intensive and therefore slow, only allows packet sniffing.

Active - An active method opens up many more avenues of attack, such as disrupting communication and sniffing the packets sent for re-establishment. In many respects this is easier than a passive attack as there is no need to calculate the LAP, UAP and clock. However, it requires transmission which adds to the complexity, and if the attack involves an attempt to connect to the master device, it can be ignored.

Pros More simple computation, often quicker, allow use of services rather

than just sniffing packets.

Cons Can be detected and therefore avoided.

2.6 Related work

There are a number of projects and products that overlap with this work, from Bluetooth device drivers and protocol stacks to tools that sniff communication between a device and its CPU. Bluez[9], the linux Bluetooth stack contains many tools and drivers to perform functions outside of the normal operation of Bluetooth. The opencores[10] project aims to create open source implementations of hardware devices such as Bluetooth baseband implementations, however the project is stale, it has not be modified since 2001, and it is still considered alpha.

There have also been a number of published Bluetooth exploits in recent years, such as the work of Yaniv Shaked in obtaining the pin exchanged at the establishment of communications[13]. The trifinite.org[14] group also maintains a list of tools and exploits for Bluetooth devices, most of which are device specific and due to flaws in the specific implementation rather than general protocol attacks.

FTS4BT[15] is a commercial Bluetooth V2.0 sniffing application that comes with specific hardware for sniffing traffic, it is expensive and is more or less a commercial, closed source, version of what is attempted here. It is aimed at debugging implementations and allows sniffing to take place within the host system so that the sniffed packets can be associated with the intended transmission data internally. The BTTracer from LeCroy also allows this functionality, but it requires the MAC address of the device to be known prior to packet sniffing.

For any of a number of attacks to be carried out there needs to be the ability to sniff packets from an arbitrary connection, this has, as yet, not been achieved, and is the aim of this work.

Chapter 3

Finding the signal

Bluetooth packets could not be sniffed and decoded until they had been received using the GNU Radio, this is not as simple as with other technologies due to the large amount of variation in the signal introduced to avoid hopping. The breakthrough in finding the signal and being able to interpret it came with the discovery of an obscure tool for manipulating the Bluetooth test modes.

3.1 Why does the signal need to be found?

Unlike most communication protocols which operate on a single frequency for a given connection, such as 802.11 and GSM, Bluetooth uses a technique called frequency hopping spread spectrum (FHSS). FHSS allows many more devices to share a set of frequency bands by moving a communication from frequency to frequency whilst in progress. This is different to 802.11 which uses multiple frequency, but each device operates on a single frequency for the entire length of the communication.

3.2 Details of the signal

The frequencies used in Bluetooth communications range from 2.402GHz to 2.480GHz inclusive at 1MHz intervals. Not all possible frequencies are used

for all communications as some are restricted in some parts of the world, but for the scope of this work all frequencies will be treated as used as this is the most common case. The frequency hops every 625us and the hopping pattern is calculated from the clock and MAC address of the master device which are only sent from master to slave at the establishment of communication.

The data is modulated using Gaussian Frequency Shift Keying (GFSK) which is a variant of Gaussian Minimum Shift Keying (GMSK) that has a specific frequency shift of 115KHz. It has a symbol rate of 1MS/s, which, when taken with the sampling rate of the GNU Radio in the 2.4GHz band, give a total of four samples per symbol (bit). When these parameters are passed to the GMSK demodulator from the GNU Radio project it is able to demodulate Bluetooth packets. These values were not given in the radio section of the Bluetooth specification document, they had to be discovered experimentally.

3.3 Experiments

An array of different experimentation methods were used in an attempt to receive and demodulate a signal generated by a Bluetooth device, these included listening on a single frequency, attempts to transmit and attempts to predict the hopping pattern of a device. The complexity came from the number of variables involved in the process, all of which had to set correctly in order to accomplish this. The first, and most obvious method for finding a Bluetooth signal is to tune to a single frequency and wait for the device to hop past. This comes with a number of problems, the first of which is knowing how to be sure that a signal has been received as opposed to a peak in noise or data from an 802.11b/g device or cordless phone.

Bluetooth data had not been previously received using the USRP and GNU Radio although the hardware was capable of receiving and demodulating the signal. The answer to this was to attempt to demodulate the signal, and identify whether it was a Bluetooth packet, but with the signal chang-

ing frequency 1600 times per second and three unknown parameters to the demodulator, this was not a simple task.

3.3.1 Random stabbing in the dark

With the device tuned to a single frequency the simplest packet was chosen to be transmitted, the ID packet; it was chosen because it is 68 bits and therefore long enough to be fairly certain that it is not noise, it is also easy to transmit with the command **hcitool scan**. The demodulator processed the signal from the USRP and wrote the demodulated data to file. A simple tool called **bin.dump** was written to take the output from the demodulator and convert it to ASCII characters '0' and '1'. The output could be searched with the standard UNIX tool **grep**.

The output from this revealed no ID packets or other packet types. The process was repeated on another frequency, and adjusted to small deviations of frequency, up to 100KHz either side of the centre frequency. When this yielded no results the various parameters given to the USRP and the demodulator were altered whilst continuing transmission on the single frequency. It became clear that some of the variables needed to be removed from the equation in order to find appropriate values for the others.

3.3.2 Test packets

Investigation of the linux Bluetooth stack (bluez) led to the discovery of a tool for controlling the CSR debug mode (bccmd) which has a rttdata1 test mode. The test transmits the same packet repeatedly on a single frequency, the packet is not whitened; this was used to transmit on a single frequency, 2421MHz was chosen to reduce noise because it was below the CPU clock signal in the host system. Radio tests transmitted on one frequency eliminated the most difficult variable in the system, the frequency hopping.

The USRP was then tuned to the frequency and the oscilloscope was used to view the signal visually. The oscilloscope was modified to take file

input, so that files of sampled signals could be viewed in order to find a file containing a single test packet. A file splitter was created to take a large file and cut it into smaller pieces for viewing with the oscilloscope, this process was refined until a file containing a single packet was produced.

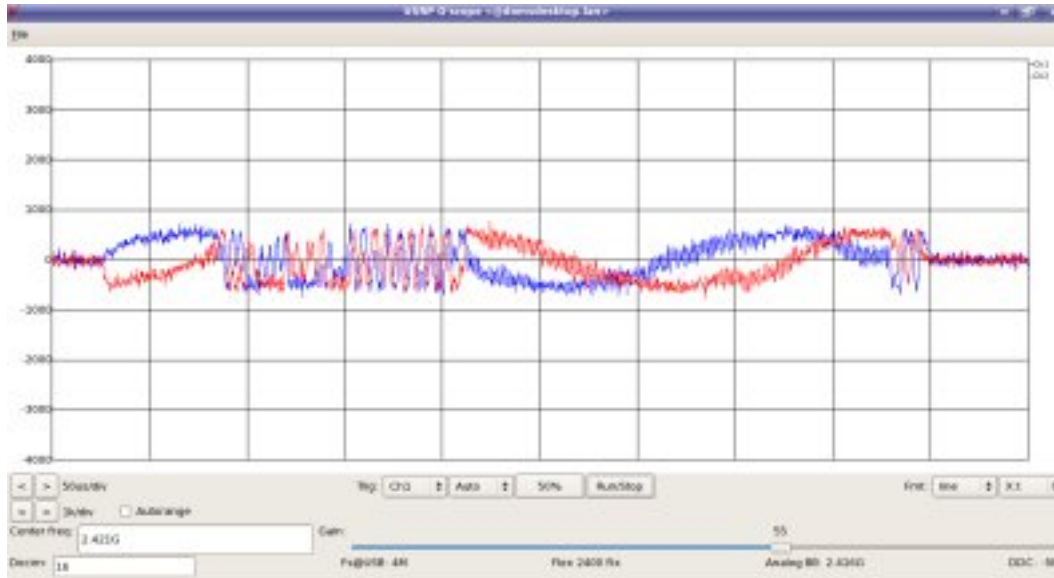


Figure 3.1: Oscilloscope with a single Bluetooth packet shown

The file containing the single test packet was passed into a python script that ran it through the GMSK demodulator, with settings as described in section 3.2 and using default values for the unknowns. Still nothing came through the demodulator, so the oscilloscope was used to calculate the length of the packet (approx 400us). The test packet consists of 366 bits, which corresponds to approximately one bit per microsecond, while the USRP samples at 4 samples per microsecond, which gives a samples per symbol value of four.

The output was dumped to file as ASCII characters using **bin.dump** so that the packet could be found by hand. Finding the packet this way was made easier by the FEC1/3 encoding on the header (see section 2.3.2) and the repetitiveness of the data.

The process of demodulating a single packet in a file is able to be scaled well to multiple packets as the demodulator works on stream data and will pass all packets through, this was tested with up to ten packets by hand, and then shown by the reception rate compared with transmission rate to be at least 80% successful. As the packets were coming through the demodulator it was considered that the correct settings had been found to receive the Bluetooth radio layer, but software was needed in order to extract useful information from these packets.

Chapter 4

Interpreting the signal

Once test packets had been found they needed to be interpreted, and data extracted from them. This was a fairly straight forward process for the test packets as they are not whitened and always carry the same LAP and UAP, so the data in them was extremely repetitive. However, the whitened packets from real transmissions were more difficult to interpret as the whitening had to be brute forced, this resulted in a list of 64 possible UAP values being given. When counted over a number of packets a set of 32 come out as the most likely, but it is not possible to determine the real value with a high enough accuracy.

4.1 Deciphering test packets

The test packets are all transmitted with the same MAC address and therefore always have the same access code, set by the same LAP. This means that the first 72 bits of each test packet are always the same, so they can be found from the output file by simple searching. Once found the header can have the FEC1/3 removed and the HEC can be calculated to check that it matches or, if required, the HEC calculation can be reversed to give the UAP as output. The type of packet can be extracted from the packet header, all of the test packets default to DH1 packet types.

The DH1 packets have a payload header which gives the length of the packet along with an ID. The payload header can be read along with the data to calculate the CRC which is initialised with the UAP. This can confirm the UAP if it is calculated from the header. The output of the program written to do this is given in Figure 4.1.

```
UAP confirmed by CRC check  
LAP:c6967e UAP:6b  
Type: DH1 Slots:1 Length:27
```

Figure 4.1: Information extracted from the test packets

4.2 Real packets

The discovery of packets from a Bluetooth device working normally, i.e. not in a debug mode, was almost accidental. The `bluetooth.dump` block was being tested with the USRP, the time for the block to stop producing output was being measure so that the delay in processing could be estimated. After the test mode was turned off three packets were received with the LAP of the Bluetooth device and varying UAPs, this is due to the whitening of the header which stopped the UAP being calculated from the HEC. These are shown in Figure 4.2.

This led to direct experimentation with two devices, using the **l2ping** tool to send repeated POLL and NULL packets between the devices. The LAP was extracted and compared to the access code to check that it was valid. Over a number of packets these were counted and when no new LAPs had been found for a time the list was printed to the screen as in Figure 4.3.

These packets were whitened, so the packet dumping functions that had been used previously did not work, in particular the method of deriving the UAP from the header and HEC does not work as the whitened data produces 64 possible packet headers, giving up to 64 possible UAPs per packet. The

```

Length:10
UAP could not be confirmed by payload
LAP:b30a5b UAP:56
Type: HV1 Slots:1 Length:10

No payload
LAP:b30a5b UAP:16
Type: POLL Slots:1 Length:0

No payload
LAP:b30a5b UAP:0f
Type: POLL Slots:1 Length:0

```

Figure 4.2: Real packets with a whitened header and payload

list of UAPs is noted for each packet, and this can be used to identify the most common UAP, which should belong to the master device.

However the voting method produced inaccurate results as the same set of 32 UAPs come out each time (shown in Figure 4.5), always gaining more votes than the number of transmitted packets.

This was put down to the same UAP coming from different versions of the header after whitening. The UAP extracting algorithm was then adapted to attempt to match on other data in the header which yielded similar results and still gave the same 32 UAPs. The same technique could be applied to other payload data, but these have variable lengths and there is no guarantee that the rest of the packet has been demodulated in order to process it.

Bluetooth LAP finder

```

Found LAP:0x452b6c  14 times
Found LAP:0xb30a5b  25 times

```

Figure 4.3: LAPs found in the within range

The list of 32 UAPs is too large to ping all possible combinations of UAP and NAP, even if filtering is done to select only the addresses in the OUI.txt file from the IEEE[16]. The OUI.txt file lists all of the upper halves of MAC addresses that have been assigned to manufacturers by the IEEE. However, if the manufacturer is taken into account a database could be built of likely and unlikely UAP and NAP pairs which may speed this process.

4.3 32 possible UAPs

The list of 32 possible UAPs that comes from brute forcing the whitening and reversing the HEC calculation is constant for the UAPs that appear in it; i.e. if one of the UAPs that appear in Figure 4.5 is set as the UAP for a device, the same set of 32 possibilities comes out. If a UAP that is not in the list is used then another set of 32 possibilities is found, these are also constant for the UAPs in the set. With 32 UAPs per set there are eight sets of possibilities. Each set has two entries for each of the most significant nibbles, and the least significant nibbles will be defined by the pairs: 4/f, 6/d, 0/b, 2/9, 7/c, 5/e, 3/8, 1/a.

These pairs also appear to come in two sets of four with 7/c, 5/e, 3/8 and 1/a in one set, and 4/f, 6/d, 0/b and 2/9 in the other. The order of these sets changes, with the pairs always remaining together. The number of occurrences of these addresses are the same, the packets that give the correct output value also give all of the other 31 values, the packets which do not give all of these 32 values give none of them, and give a random set of other results, it is assumed that there are bit errors in these packets, and they are disregarded.

At this point the validity of the HEC reversal and whitening removal are questionable, but the testing in section 5.6.2 shows that both conform to the specification. This leaves the option that Bluetooth was designed this way, either it was a mistake and it is a flaw that allows the sniffing of a single packet to reveal four bytes of the MAC address to within 32 possibilities, or

the whitening algorithm was designed to work with the HEC algorithm to give a set of 32 possibilities rather than just the one correct one.

The list of 32 UAP candidates could be reduced depending on their packet type, for example a DH1 packet has a variable length payload followed by a CRC which could be checked, an EV3 packet has a fixed payload length (unknown at time of sniffing) which is also followed by a CRC. The packet types are also often used in groups, i.e. the HV packets are used together, as are the EV packets, and the DH and DM packets appear in the same connections, this makes it possible to filter the packets on the connection ID, and type of packets being used in that connection. The generic control packets (NULL, POLL, etc) are added to all of the groups as they are used with each connection type.

This method of filtering produces results with the correct value often, but not always, having the maximum number of votes, or being close to it, but it is almost never alone at the top of the group. This situation remains the same as the number of packets is increased, so that 5000 packets has no advantage over 100 packets. The extra filtering does however reduce the likely options, cutting off either end of the list; these results are shown in Figure 4.4. They also have the most common LT_ADDR shown for each UAP, the chances are high that the correct result has an LT_ADDR of 1 because there are only two devices in the piconet, if information about a piconet is known, then this can be used to make an intelligent guess at the identity of the correct UAP. In this case the possibilities are narrowed from 256 to 4.

```
40 -> 53 votes -> LT_ADDR 5
4b -> 56 votes -> LT_ADDR 5
52 -> 56 votes -> LT_ADDR 3
59 -> 54 votes -> LT_ADDR 3
60 -> 56 votes -> LT_ADDR 7
6b -> 54 votes -> LT_ADDR 7
72 -> 54 votes -> LT_ADDR 1
79 -> 56 votes -> LT_ADDR 1
84 -> 52 votes -> LT_ADDR 7
8f -> 54 votes -> LT_ADDR 7
96 -> 54 votes -> LT_ADDR 1
9d -> 53 votes -> LT_ADDR 1
a4 -> 54 votes -> LT_ADDR 5
af -> 53 votes -> LT_ADDR 5
b6 -> 51 votes -> LT_ADDR 3
bd -> 54 votes -> LT_ADDR 3
```

Figure 4.4: Filtered output of the Bluetooth UAP finder, 72 is the correct value, it also has an LT_ADDR of 1

Possible UAPs within 20% of max value
max value=46

00-	> 38votes
0b-	> 42votes
12-	> 38votes
19-	> 46votes < - - - -Highest voted value
20-	> 39votes
2b-	> 42votes
32-	> 37votes
39-	> 46votes
40-	> 43votes
4b-	> 38votes
52-	> 42votes
59-	> 38votes
60-	> 42votes
6b-	> 37votes
72-	> 42votes < - - - -Correct value
79-	> 39votes
84-	> 40votes
8f-	> 43votes
96-	> 39votes
9d-	> 41votes
a4-	> 39votes
af-	> 41votes
b6-	> 41votes
bd-	> 41votes
c4-	> 42votes
cf-	> 37votes
d6-	> 44votes
dd-	> 39votes
e4-	> 44votes
ef-	> 38votes
f6-	> 42votes
fd-	> 38votes

Figure 4.5: Output of the Bluetooth UAP finder, the correct value is 72, but the most votes go to 19

Chapter 5

Implemented functions

The finding and interpreting of a signal is of little use unless it can be reliably repeated, a detailed description of the setup used follows. This starts with the physical layout of the hardware components, follows with a description of the scripts and software functions used to convert an analogue signal to digital data, and finishes with tools to extract information from the data.

5.1 Experimental setup

5.1.1 Physical layout

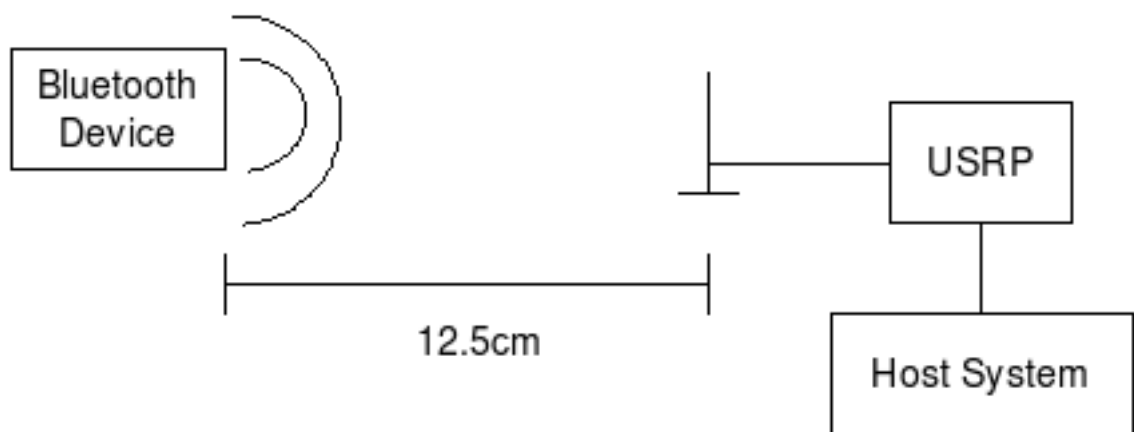


Figure 5.1: Hardware setup for receiving Bluetooth

The devices were arranged as in the Figure 5.1.1, with a minimum of 12.5cm (one full wavelength at 2.4GHz) between the Bluetooth USB device and the antenna for the radio. The antenna must be pointed away from the host system as many CPUs have a clock signal that is in the 2.4GHz frequency band, and this causes a large amount of noise and interference.

5.1.2 Data flow of the signal

Figure 5.1.2 shows the stages that the signal received by the radio must pass through in order to be processed. These modules are part of the GNU Radio framework, from the USRP hardware, to the software filters and demodulators that come with the software framework. The Bluetooth module was written in order to process the binary data that comes from the demodulator.

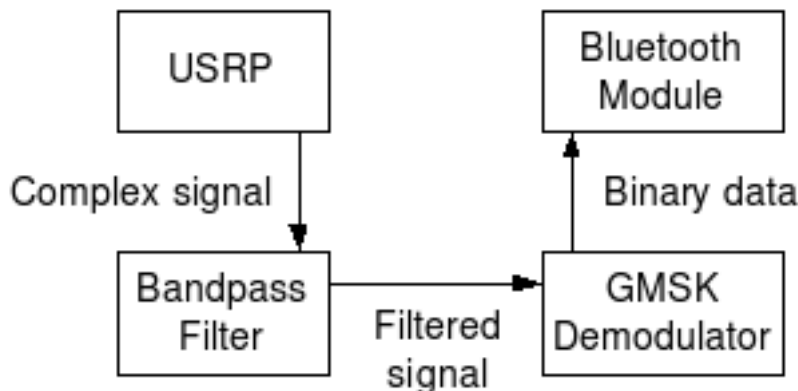


Figure 5.2: Flow of data within the GNU Radio software

The USRP control, filters and GMSK demodulator are part of the GNU Radio framework, however the Bluetooth specific block was written for this work, and therefore the contents of it is described in greater depth below.

5.2 GNU Radio gr-bluetooth block

The various methods and functions that have been implemented have been incorporated into a GNU Radio block which, following the GNU Radio convention, is named gr-bluetooth. The structure of the source and build files

is taken from the example block (gr-howto). The Bluetooth block has four modules: dump, LAP, UAP and sniffer, which are described below.

5.2.1 Modules

Bluetooth.dump Dumps packet information to stdout based on packets found. Gives the LAP, UAP, and length of the packets found. It does not work on whitened data, only on the CSR test packets. It uses `acgen()`, `sniffac()`, `unfec13()`

Bluetooth.LAP Looks at traffic to sniff for access codes, then checks that they match the LAP that created them. Gives a list of LAPs found, exits when it only finds LAPs that it has already checked. Uses `acgen()` and `sniffac()`.

Bluetooth.UAP Takes an argument of a given LAP and only examines packets from the piconet that uses the associated access code. This will give packets that have been sent or received in the piconet that we wish to listen to. From these packets the UAP of the master device is determined. It uses `acgen()`, `sniffac()`, `unfec13()`, `unwhiten()` and `UAP_from_hec()`

Bluetooth.sniffer Bluetooth sniffer is intended to be a full implementation of the above functions, i.e. it will find packets within a stream given an unknown LAP, identify the LAP and then attempt to identify the UAP from the packet headers and possibly other packet data. Once this data is found it should sniff packets to confirm. It can handle multiple packet types and perform CRC check where required. However, due to the problems in finding the UAP of other devices it is not fully functional, and largely untested. It uses all of the common functions including `crcgen()` and `unfec23()`. The flow of data through the sniffer is shown in the Figure 5.3.

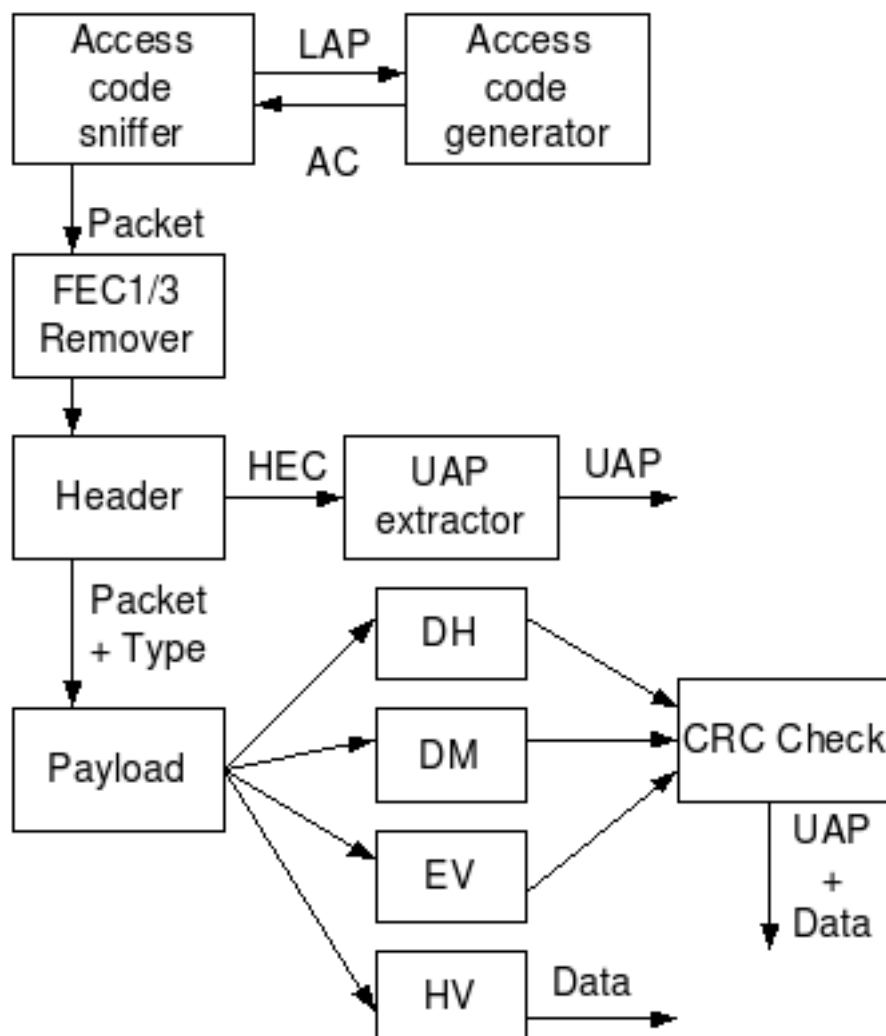


Figure 5.3: Function involved in processing packets

5.3 How the modules work

All of the modules written are based around the same functions, the most important of which are described here. This is shown in order the that the signal is passed to them, as in Figure 5.3.

5.3.1 Access Code

acgen() creates an access code (as described in 2.3.1, including preamble, syncword and trailer, from a given LAP. It uses the codeword function, written by Dr Robert Morelos-Zaragoza of San Jose State University, to produce the codeword, then combines this with the LAP and preamble/trailer to return data to the calling function.

sniffac() works through a stream of data to find the **preamble** and **trailer** of an access code at the correct distance apart. It then takes the portion that represents the **LAP** and uses **acgen()** to calculate the **codeword**. The created access code is then compared to the one that was sniffed, and the offset of the packet in the stream is returned if it matches.

101010110111110011000010110110011001100111111001101001011000111100101010

5.3.2 Header

unfec13() removes the FEC 1/3 encoding from the packet header and/or payload, by averaging sets of three bits. Input and output are both in one-LSB-per-byte format.

unwhiten_header() is used to unwhiten 18 bits based on a given clock value, this is implemented using two arrays, one contains 64 entries and is a look up table for the index into the second. The second array contains 127 bits of whitening data to XOR with the packet header, the 18 bits from the index onwards are XORed with the received header to reveal the intended header data.

hecgen() implements exactly the LFSR that creates the HEC from the header. It takes the stream of data and the UAP of the master device in and produces the HEC for the header of the packet.

UAP_from_hec() reverses the **hecgen()** by taking the header, complete with

HEC, and running the process backwards starting with the HEC and working back to the UAP. It works almost exactly as the `hecgen()` does, but it was implemented directly from the specification in order to use the two functions to test each other.

5.3.3 Payload

`crcgen()` is similar to `hecgen()` but it implements the LFSR for creating the CRC, and takes the length of the packet as an argument because it is applied to variable length data.

`unfec23()` removes the FEC 2/3 encoding from the payload if required. It takes the payload ten bits at a time and drops the 5bit FEC code. This was chosen as the best way to deal with the FEC 2/3 packets, because the payload is not of concern at this stage of the work. Also, if there is a CRC then this will show signs of errors in transmission, and the FEC codes can then be explored. This increases the speed at which the system operates when the reception is good and errors are infrequent. Input and output are both in one-LSB-per-byte format.

5.4 GNU Radio scripts

There are four scripts that accompany the above functions that are used to execute them from the command line. `bluetooth_LAP_sniffer.py` takes a frequency and gain, it runs the `bluetooth.LAP` module to find the LAPs being used within range. `bluetooth_UAP_sniffer.py` takes a frequency, gain, LAP and a number of packets and passes this data to the `bluetooth.UAP` module.

`bluetooth_dump.py` and `bluetooth_sniffer.py` are more complete applications, `bluetooth_dump.py` reads from the USRP on a single frequency and dumps all received packets to stdout, it does not handle any whitening. It is useful for testing that the setup is correct but using it with the CSR test packets. `bluetooth_sniffer.py` is an attempt at a full implementation of a

packet sniffer on a single frequency, it runs the `bluetooth.sniffer` module.

5.5 **bccmd**

One of the tools that comes as part of the `bluez` utilities package is `bccmd`. It is mostly used to access the debug modes of the Bluetooth devices that were designed by Cambridge Silicon Radio. The debug mode allows the device to be locked to a single frequency and then forced to transmit simple test packets.

The packet type, length and content of the test packets are configurable in the debug mode on the CSR chips, but this was unimplemented in **`bccmd`** because so far it had not been required. A packet config function was added, and the ability to select the specific radio test was added to the existing radio test function.

5.6 Testing

The methods of testing the various functions involved in extracting information from the data stream produced by the GNU Radio are shown below. The results of these tests are given in Appendix C

5.6.1 Access code

acgen(), the access code generator, was tested using sample data given in part G, section 3 of the Bluetooth specification v1.2[7]. The **acgen()** function was put into a separate C file along with all functions that it calls, the LAPs that are given in the specification were passed to it in turn, and the output dumped to stdout. This was piped to file, which was compared to the sample data using **diff**.

sniff_ac() was tested with the USRP data after packets had been found by the packet finder, the sniffer was able to be tested on files that were known to contain a packet as well as data from the USRP.

unfec13() was tested using sample data, and by review of the code as it is a simple function.

5.6.2 Header

unwhiten_header(), the packet whitening remover was tested using the sample data given in part G, section 7 of the Bluetooth specification v1.2[7]. The data was put into a file, and the packet whitener was initialised with the value 0x3f (the first in the list of sample data), it was then run until the shift register had the same value in again, meaning one full cycle of values had been generated. As the only input to the packet whitening algorithm is the clock at the start, after this no input is given, so this test represents every possible state of the shift register.

hecgen() was tested with the data in the part G, section 4 of the speci-

fication. It was also tested by `UAP_from_hec()`, as both were implemented from the specification details in order to test them against one another.

`UAP_from_hec()` was tested with the known to be working `hecgen()`, the data from one was passed to the other to see if the expected UAP could be derived. The USRP captured data was used to test that the correct UAP was produced.

5.6.3 Payload

`crcgen()` was also tested against the data in the spec, although this was minimal, so the data captured by the USRP from the CSR test packets was also used.

`unfec23()` was tested using sample data, and by review of the code as it is a simple function. If it was required to calculate the parity bits it would be tested against the data in part G section 8 of the spec.

Chapter 6

Evaluation

The work in this project is difficult to evaluate in the way that most networking research is evaluated because it does not deliver a numerical improvement or a service that can be statistically analysed. The result of the work is that packets are sniffed, and information is extracted from them in real time. This provides a significant advance in the field of Bluetooth hacking as this has not been possible up to this point, but it is not an outcome that can be shown in a graph or a bar chart.

6.1 The USRP's role in Bluetooth hacking

One of the larger outcomes of this work has been to show that the USRP device, along with the GNU Radio software is able to read and demodulate packets from a connection between two Bluetooth devices. This alone assures its use in Bluetooth future Bluetooth hacking based on the radio layer, but the flexibility of the software to allow this to be coupled with information extraction in one process means that it could be used as the basis for a single tool to perform an attack.

The only drawback that the hardware may have is a potential show stopper. The length of time it takes the device to re-tune to a frequency is roughly one third of the length of a packet, which would result in packets

being missed as the radio attempted to tune to the frequency of the next hop. This does not have to be a problem though as there is the possibility of installing a second daughterboard, and hence a second receiver into a USRP device, giving two radios which can alternate their hopping. The use of two radios in one device would allow each `dubdevice` 625us to tune, which is ample time.

6.2 Exploiting the Bluetooth protocol

Now that the place of the USRP device in Bluetooth hacking has been assured the feasibility of attacking the protocol at the radio and baseband layers needs to be evaluated.

6.2.1 Finding the MAC

The three least significant bytes of the MAC address of the master device can be found simply from reception of a single packet, and this may be confirmed with subsequent packets. The NAP cannot be found from standard packets, it is only transmitted in the FHS packet but if the UAP is known, then the `OUI.txt` file from the IEEE can be used to narrow the search down to tens rather than thousands of possibilities.

The real trick is in finding the UAP of a device, this can be done by reversing the error check on the header. This is an extremely useful technique as it means that a specific packet does not need to be waited for or the connection disrupted in order to get the UAP. However, the process becomes more difficult when the packet whitening comes into the equation as this can only be removed with brute force, which provides 64 possible UAP candidates. Filtering, and some knowledge of the connection, helps to reduce this list of candidates to less than 10, which gives a much higher chance of snooping a connection.

6.2.2 Finding the clock

There are a number of possible ways of finding the clock signal which were investigated, but until the MAC address can be found they will remain entirely theoretical. These either involve reversing the hopping algorithm, or learning the full address of the device and attempting to ping it, which would result in it revealing the hopping pattern. The more reliable option is to guess the NAP of the device as there are roughly thirty NAPs per UAP in the OUI.txt file available from the IEEE. This will be a lot quicker than trying to reverse the hopping pattern.

However, part of the clock signal can be derived by removing the whitening from a packet, and this will reduce the amount of the clock signal to be discovered. Also, this method is less invasive than pinging devices, and therefore can still sniff packets from a device that ignores l2ping requests. There does not seem to be a method for calculating the rest of the clock from transmitted packets without receiving an FHS packet.

It may be possible to reverse the hopping selection algorithm without interpreting any packets, using fast Fourier transforms and observations of the frequency range. This would reveal the MAC address and clock of the master device so that the hopping pattern could be derived and followed, gaining the advantage of not having to deal with extracting the UAP from whitened packets. A patent is held on this technique by Cognio Inc[12]. This would require multiple USRP devices, or even custom hardware with a very wide bandwidth, which reduces the possibility of it being carried out by anyone but a well funded attacker.

6.2.3 Putting it all together

As the entire MAC could not be found, and only six bits of the clock could be estimated, but not known, implementing hopping was not possible, but assuming that this information had been found the attacks that could now be perpetrated need to be investigated. A number of possibilities are given.

Force disconnect

At the logical link layer of the protocol the devices see the baseband layer as a fixed communication link. The devices need a method for disconnecting when they are done with the link, and this is done with a disconnect packet. If a disconnect packet is received by a device it will drop the link described in the packet.

This will force the connection to be re-established if either device still wishes to send data, meaning that the FHS packet will be sent from the master device to the slave. This can then be sniffed by the USRP and the data can then be extracted, giving the MAC address and clock values.

Following the reception of the MAC and clock values the hopping pattern could be followed and the data sniffed. Whitening would also not be a problem as the clock value is known.

Sniff data

Sniffing data as a purely passive attack is a preferred option as it will not rouse suspicion and is therefore very difficult to detect and protect against. This would work the same way that the “Force disconnect” attack does, but instead of forcing the disconnect to view an FHS packet, the connection is observed passively to calculate the MAC and clock value. This could possibly be achieved by viewing the frequency hopping pattern using and reversing the hopping selection algorithm to get the MAC and clock.

Man in the middle

A man in the middle attack is less likely to be successful as the devices communication will often be within range of one another, and the interference from the hopping patterns would need to be mitigated somehow. This attack would involve forcing disconnect, and then appearing as the two devices to each other, this is complicated, and the timing may be critical. It is less likely to succeed than the previous suggestions.

Chapter 7

Conclusions

7.1 Have the aims been met?

The aims set out for this work were to implement packet sniffing and injecting for Bluetooth using the GNU Radio and USRP device. To a certain extent sniffing packets is now possible, but some of the functionality required to fully sniff communication between two devices is not available yet. This is due to the constraints of the USRP hardware when frequency hopping and the complexity of extracting the UAP from the packet; the further work section includes suggestions as to how to accomplish these.

7.2 Why is this not a full implementation?

7.2.1 Finding the signal

The decision was taken fairly early in the project to avoid a direct implementation of Bluetooth using the GNU Radio, as this was not going to be of much use. Anyone who wishes to use Bluetooth for communication can simply use a \$10 USB device rather than a \$1000 software defined radio, but for sniffing packets a standard device may not suffice.

The added complexity in finding, demodulating and interpreting a Bluetooth signal was a large hurdle to finding and exploiting flaws in the security

of Bluetooth, although it was a task that has helped a number of people to investigate the protocol at the radio and baseband layers, and it will hopefully lead to future developments in the field. Most of the theoretical exploits for Bluetooth, that have been release to date, cannot be practically applied without the ability to sniff data at the baseband level. All further work, described in Chapter 8 , is with a view to implementing one of these attacks.

7.2.2 Lack of frequency hopping

The USRP daughterboards cannot tune to a frequency quickly enough to keep up with the hopping pattern of Bluetooth, which hops to a new frequency every 625us in order to spread the transmissions[7]. The RFX-2400 daughterboard to the USRP takes around 200us to settle to a frequency when a tune command is issued[11].

This means that roughly the first third of a transmission slot will be missed as the radio tunes to the frequency. This factor could be reduced by using the transmit and receive paths of the daughterboard or adding a second board; although this will double the data transferred over USB, the bus should be able to handle it as it presently uses 4Mb of bandwidth but can support up to 32Mb, which would be plenty for two daughterboards with two receivers on each. The tune time is also likely to be less if the frequency is within a few MHz of the current setting, but this is untested.

7.3 Software written

The software written and described in this document is a proof of concept, it shows that the USRP device can sniff packets, and they can be demodulated. It also shows that information, such as the lower half of the MAC, and six bits of the clock, can be extracted from the demodulated packets.

It would be useful to have a well designed object oriented version of the software which could handle multiple connections and give stats about the

devices operating in range of the USRP. This would be a valuable extension to the project to enable it to be reasonably used as a tool for attacking Bluetooth communications, but it was not the focus of this project.

The project sets out to expose the security flaws of Bluetooth and show that it is possible to sniff the packets, as it is with any wireless protocol. This task needed to be done in order to enable a practical attack to be performed.

7.4 Where does this leave Bluetooth, GNU Radio and the USRP?

This work brings Bluetooth hacking to a new position where it is closer than ever before to a practical attack on the protocol itself, rather than exploiting the flaws of particular implementations. The USRP hardware is not out of reach for the average attacker, and this means that it is a very real threat.

The GNU Radio project will now have the code for this project contributed to their resources, so that a full implementation and a practical attack can be attempted in an open source environment, giving any attacker the same advantages. The research will hopefully contribute to the view that no wireless protocol is safe from packet sniffing, and frequency hopping spread spectrum is not a security technique, it is merely a method for reducing interference.

Bluetooth is an extremely prevalent technology, and it will continue to be used for more and more data transfer, but with new versions will come new security measures, meaning that sniffing packets is only the first step to exploiting protocols at higher levels of the stack.

Chapter 8

Further work

Now that the GNU Radio can receive packets from a Bluetooth connection there are only a small number of functions to implement in order to perform an attack. These are finding the clock value of a device, which can be done in a number of ways described below. With the clock found, an attack should be relatively simple to carry out.

8.1 Gathering more data from sniffed packets

A single packet can currently yield the LAP and, if the packet has a CRC, the UAP and six bits of the clock; but there may be more information that can be gathered. An FHS packet contains all of the information needed to follow the hopping pattern and remove the whitening of the packets, but it is only ever sent at the time of a connection establishment.

Once the UAP has been discovered then packets can be read, and the clock change between them noted, thus giving the time between packets received so that their place in the communication can be calculated. It will also allow a more accurate guess at the clock input to remove the whitening from the data, which should speed up the interpretation of the signal.

8.2 Hopping

Hopping is the only major hurdle left to a successful attack against a device, and for this the clock signal is needed. Once the clock signal and address have been found then the hopping pattern for the device can be discovered. The hopping pattern can be calculated from only the UAP and LAP parts of the address, and all 27 bits of the clock signal. The algorithm that determines the frequency of the next hop is described in section 2.6.2 of the Bluetooth spec[7].

8.3 Using commodity hardware

At the time of writing the GNU Radio hardware costs around \$700 plus an additional \$275 per RFX-2400 daughterboard[17]. This is not too much for anyone who wishes to listen to the Bluetooth communications of others, especially if there is financial gain to be had. However, the security of the technology cannot really be said to have been compromised until communications can be sniffed with commodity hardware; this requires an attack using only the \$3 Bluetooth chip built into a device. In April 2007 a CSR based device was reportedly turned into a Bluetooth sniffer using firmware taken from a commercial sniffing tool[18]. However this is unconfirmed and the capabilities are not known; if this can provide all of the functions that are required to snoop a communication then it could be used to replace the GNU Radio for an attack.

8.4 Mounting an attack

The overall goal of this, and all Bluetooth security research, is a successful attack mounted against arbitrary devices, preferably using commodity hardware. However, all of the above problems need to be solved before this can happen. The USRP seems well suited to the task of sniffing packets, but the precise timing required for transmission and reception on the same device may prove impossible with the current GNU Radio framework. One option

to avoid this problem is to use the USRP to sniff packets and discover the MAC/clock values, and then use the bccmd tool to configure a device with these parameters so that it will follow the hopping pattern and be able to snoop the entire communication.

Appendix A

System manual

This document describes the way in which the tools that have been written work. It is intended for anyone who wishes to modify or use any of the tools. For a guide to running the tools, see the User Guide.

A.1 Command line tools

A.1.1 `bccmd`

The modifications to `bccmd` were to add a "radiotest" function, which was mostly a direct copy of "rttxdata1" but instead of hard setting the TestID value, it is taken from the third command line argument. A "pktconfig" mode was added to allow the length and type of the test packets to be altered, this is based on the radiotests with a TestID of 23.

A.1.2 `packet_finder`

The `packet_finder` has a hardcoded LAP, which is used to calculate an access code. This access code is then used to compare to the input stream of data. The `find_ac` method allows for up to five errors in the access code to allow for transmission and demodulation errors. This was designed to only find the DH1 packets from the CSR test mode, so it does not deal with any payload types other than

DH1, and does not handle with whitened data.

A.1.3 `packet_sniffer`

The **packet_sniffer** is a modified version of the **packet_finder**. In the `find_ac` function it takes the LAP from the access code, then creates an access code based on the LAP, before checking that this matches what was read in. Unlike `packet_finder` it does not allow any errors. This does not seem to cause any problems with reception as a large number of the test packets get through. The `packet_sniffer` is not intended to be used for anything but a proof of concept of the methods to be used in the GNU Radio block listed below.

A.2 GNU Radio Bluetooth (`gr-bluetooth`)

A.2.1 `sniffer`

The **sniffer** module is a version of the `packet_sniffer` above modified to fit within a GNU Radio flow graph. The main function is replaced by the standard work function of GNU Radio's `gr_sync_block`.

A.2.2 `bluetooth_LAP_sniffer.py`

The **LAP** module is designed to only snoop LAPs of the devices working in range, it exits after a time if no new LAPs have been found. It will not exit if no LAPs have been found.

A.2.3 `bluetooth_sniffer.py`

bluetooth_sniffer.py combines the USRP input, GMSK demodulator and **sniffer** listed above. Acts as a packet sniffer for Bluetooth packets (although doesn't seem work for anything but the CSR test packets at present). It is mainly used to

test the **sniffer**, but also to be more simple to execute than **packet_sniffer**.

A.2.4 **file_sniffer.py**

file_sniffer.py is similar to **bluetooth_sniffer.py** except that it does not use the USRP for input, it is useful when checking sampled files from the radio in order to compare the output of **sniffer** to **packet_sniffer**. Unlike **packet_sniffer** it takes complex files, and therefore passes the files through the demodulator before processing.

Appendix B

User manual

A guide to the tools that have been written or modified whilst attempting to receive Bluetooth packets with USRP hardware. For a more in depth description of how the tools work, see the System Guide.

B.1 Command line tools

B.1.1 `bccmd`

bccmd is a tool that comes as part of the bluez linux Bluetooth stack. It allows some fairly direct control over the Cambridge Silicon Radio chips used in a number of Bluetooth devices. One of the options is a radio test mode, this sends one of the transmission tests repeatedly. There are alternative transmission tests, and these can be useful to access, so this has been added. A packet confing mode has been to allow the length and type of the test packets to be changed.

The extra options are used as follows:

bccmd radiotest freq level TestID

Where:

freq is the frequency to broadcast the test packets on in MHz, between 2401 and 2479.

level is the volume for transmission, values between 0x0000 and 0x003f are recommended.

TestID is the numerical ID of the test, 4, 6 and 7 are useful transmission tests.

bccmd pktconfig type length

Where:

type is the identifier of the packet type, e.g. 3 is DM1, 4 is DH1, 7 is EV3.

length is the length in bytes of the data in the payload.

B.1.2 packet_finder

packet_finder is a tool to print out statistics of all packets in a data stream for a given LAP. This allows us to check that the access code is correct before attempting to gather information from the rest of the packet. As input it takes a binary file packed 1 bit per byte (the least significant bit of the byte), this is the standard output of the GNUradio GMSK demodulator.

The format of the command is: **./packet_finder inputfile**

The program will exit when it reaches the end of the file. If the start of a packet is found, but the end is cut off, an exit code of 1 will be given.

B.1.3 packet_sniffer

Whilst **packet_finder** will find all packets with an access code based on the given LAP, we may not know the LAP of the device we wish to sniff packets from. This is where **packet_sniffer** comes in. It scans through the input stream to find the preamble and trailer of an access code, and then checks that the LAP portion creates the codeword portion of the syncword.

As input it takes a binary file packed 1 bit per byte (the least significant bit of the byte), this is the standard output of the GNUradio GMSK demodulator.

The format of the command is: **./packet_sniffer inputfile**

The program will exit when it reaches the end of the file. If the start of a packet is found, but the end is cut off, an exit code of 1 will be given.

B.2 GNU Radio Bluetooth (gr-bluetooth)

B.2.1 sniffer

The Bluetooth packet **sniffer** block for GNU Radio is based on the **packet_sniffer** command line tool. It takes demodulated data from the GMSK block and searches for possible access codes, then it attempts to gather as much information as possible about each packet before printing it to the screen.

B.2.2 bluetooth_sniffer.py

bluetooth_sniffer.py is a script that links the USRP input to the GMSK demodulator and then passes this to Bluetooth sniffer block.

It is used as follows:

```
gr-bluetooth/examples/bluetooth_sniffer.py -f freq -g gain
```

Where **freq** is the frequency to listen on, and **gain** is the gain to be applied to the incoming signal.

B.2.3 file_sniffer.py

file_sniffer.py is a script to pipe a file into the sniffer block instead of taking the input from the radio device and the demodulator. This is useful for debugging when used in conjunction with a file that is known to contain packets as it removes errors that come with noise.

It is used as follows:

```
gr-bluetooth/examples/file_sniffer.py inputfile
```

B.3 Frequency / gain

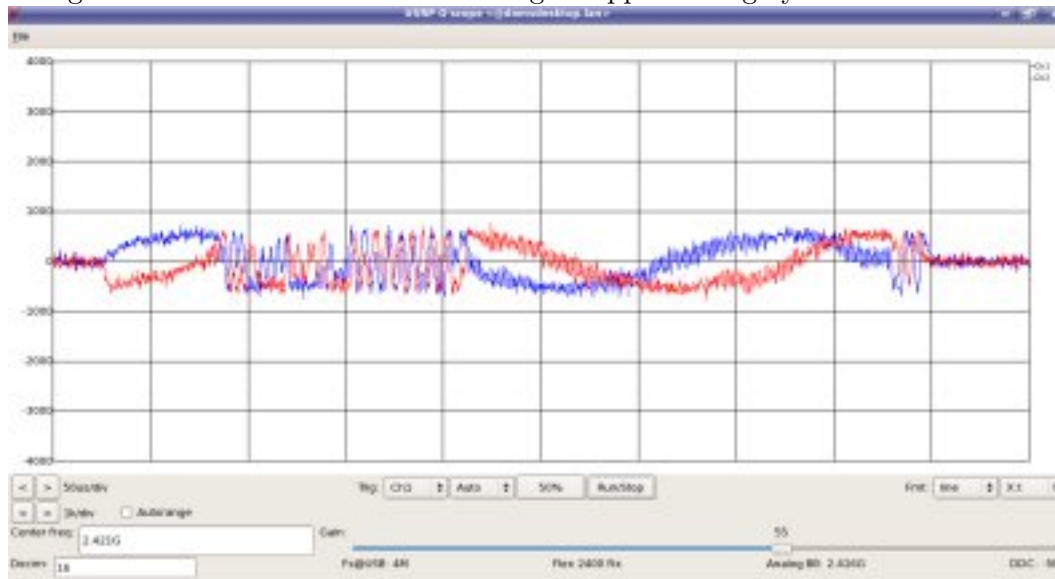
B.3.1 Frequency

All values of frequency are valid between 2401MHz and 2479MHz, the transmissions are supposed to be at 1MHz intervals, but there may be inaccuracy and drift,

this can be identified using the `gnuradio-examples/python/usrp/usrp_fft.py` which displays a plot of the frequency distribution of the signal.

B.3.2 Gain

The gain needs to be set such that the signal appears roughly like this:



With the settings at 50us and 1k per division, the signal should stretch between the +1000 and -1000 lines.

Appendix C

Test results

C.1 Packet codes and error checks

C.1.1 `acgen()`

All of the 130 LAPs from volume 2, part G, section 3 of the Bluetooth specification v1.2[7] were given as input to the `acgen()` function. The expected output was then taken from the same section of the spec, copied into a file. The output from `acgen_test` was formatted to the same style as the output in the spec, and was saved to file. The two files were then compared with the `diff` tool and no differences were found. All 130 access codes were generated correctly.

C.1.2 `hecgen()`

The specification gives 20 sample packet headers with HEC, these were used to test the `hecgen()` function. The function was also tested by checking the HEC on the CSR test packets when they were received.

C.1.3 `crcgen()`

The specification only contains one piece of sample data for testing CRC calculations. This was tested, and the `crcgen_test` program produced the correct output. However this was no conclusive enough, and further testing

was provided by the test packets sent from the CSR device. The packets had a fixed UAP which was known, so the `crcgen()` was included in the code to extract data from them; it was used to check that the CRC present at the end of the packet was correct given the data in the payload. It was able to check all of the packets presented to it, and identified correct packets, and also rejected packets with errors present.

C.2 Packet sniffing and data extraction

C.2.1 `sniff_ac()`

`sniff_ac()` is able to find packets in a stream of data, finding all packets that have no errors in the access code. This could be extended to allow for a small number of bit errors in the access code, but the speed of execution is considered more important.

C.2.2 `UAP_from_hec()`

All HECs produced by `hecgen()` were able to be reversed, given the appropriate header data, to reveal the correct UAP for the packet. The correct UAP was given when the CSR test packets were passed to the function, which confirmed the test.

C.2.3 `unwhiten()`

The unwhitening tool was implemented twice, once as the specification showed, with an LFSR, this was then tested against the sample data given. The second version was created directly from the sample data as a lookup table, this was tested against the first implementation to ensure that it produced the correct results.

Appendix D

Interim report

D.1 Progress made to date

Most of the effort so far has gone into reading the specifications of Bluetooth and the Universal Software Radio Peripheral (USRP). The Bluetooth protocol involves some interesting radio techniques in order to minimise interference in an already crowded frequency band (2.4GHz), these techniques need to be implemented using the USRP. Some of them have already been implemented, others have not, although it is possible that these are constrained by the current hardware in the USRP.

The software of choice for the USRP is the GNU Radio project, an open source software defined radio package closely related to the USRP device. The processing of signals is mostly performed by c++ "blocks" which are connected together in a graph by python. This has involved getting to grips with python and c++, but this has not been a problem.

The USRP allowed a chance to experiment with receiving signals and processing them. This was more difficult than expected due to lack of experience with radio technologies, but it is now possible to record a signal and pass it to the demodulator using the GNU Radio framework.

This signal currently includes a lot of noise which needs to be eliminated

using filters supplied by the GNU Radio software. The Bluetooth specification also states that certain packet types have repetition, and this can be used to calculate the correct demodulation of the signal.

D.2 Further work

There's a lot of work still to be done, most of which will involve processing the packets from the demodulator stage onwards. This can be achieved within the context of the flowgraph using modules written in c++.

- The next stage is to identify the Bluetooth packets being received by the software, and identify/classify them (expected completion: End of January). I will most likely write a Bluetooth version of TCP dump to do this. This should be able to identify the packets used to establish a connection:
 - ID packets
 - FHS packets
 - NULL packets
 - POLL packets
- Following the successful reception of Bluetooth packets, transmission of packets is the obvious next step, starting with the ID packet (Reading week).
- The USRP device is unable to change frequencies quickly enough to synchronise with a device's hopping pattern, but the protocol allows the master device to set the hopping pattern. This will make it possible to communicate with a Bluetooth device, as long as the device allows itself to be the slave in the connection (End of March, due to exams).
- The inability to hop frequencies quickly will most likely mean that connections between other devices cannot be eavesdropped, but this may be possible with future versions of the hardware. For now the methods for using the USRP to sniff packets will be discussed in the final report, but they're unlikely to be implemented (Mostly involves the writeup, due in April).

Appendix E

Code

E.1 Access code

E.1.1 codeword()

```
1  uint8_t *bluetooth_UAP::codeword(uint8_t *data, int length, int k)
2  {
3      int i, j;
4      uint8_t *g, *cw, feedback;
5
6      g = (uint8_t *) malloc(35);
7      g[0] = 1;
8      g[1] = 0;
9      g[2] = 0;
10     g[3] = 1;
11     g[4] = 0;
12     g[5] = 1;
13     g[6] = 0;
14     g[7] = 1;
15     g[8] = 1;
16     g[9] = 0;
17     g[10] = 1;
```

```
18     g[11] = 1;
19     g[12] = 1;
20     g[13] = 1;
21     g[14] = 0;
22     g[15] = 0;
23     g[16] = 1;
24     g[17] = 0;
25     g[18] = 0;
26     g[19] = 0;
27     g[20] = 1;
28     g[21] = 1;
29     g[22] = 1;
30     g[23] = 0;
31     g[24] = 1;
32     g[25] = 0;
33     g[26] = 1;
34     g[27] = 0;
35     g[28] = 0;
36     g[29] = 0;
37     g[30] = 0;
38     g[31] = 1;
39     g[32] = 1;
40     g[33] = 0;
41     g[34] = 1;
42
43     cw = (uint8_t *) malloc(34);
44
45     /* This section written by
46        Dr Robert Morelos-Zaragoza
47        of San Jose State University */
48
49     for (i = 0; i < length - k; i++)
50         cw[i] = 0;
```

```

51     for (i = k - 1; i >= 0; i--) {
52         feedback = data[i] ^ cw[length - k - 1];
53         if (feedback != 0) {
54             for (j = length - k - 1; j > 0; j--)
55                 if (g[j] != 0)
56                     cw[j] = cw[j - 1] ^ feedback;
57             else
58                 cw[j] = cw[j - 1];
59             cw[0] = g[0] && feedback;
60         } else {
61             for (j = length - k - 1; j > 0; j--)
62                 cw[j] = cw[j - 1];
63             cw[0] = 0;
64         }
65     }
66     free(g);
67     return cw;
68 }

```

E.1.2 acgen()

```

1  /* Endianness - LAP is MSB first */
2  uint8_t *bluetooth_UAP::acgen(int LAP)
3  {
4      uint8_t *retval, *pn, count, *cw, *data;
5      retval = (uint8_t *) malloc(9);
6      pn = (uint8_t *) malloc(9);
7      data = (uint8_t *) malloc(30);
8
9      LAP = reverse((LAP & 0xff0000)>>16) |
10         (reverse((LAP & 0x00ff00)>>8)<<8) |
11         (reverse(LAP & 0x0000ff)<<16);
12
13     retval[4] = (LAP & 0xc00000)>>22;

```

```

14     retval[5] = (LAP & 0x3fc000)>>14;
15     retval[6] = (LAP & 0x003fc0)>>6;
16     retval[7] = (LAP & 0x00003f)<<2;
17
18     /* Trailer */
19     if(LAP & 0x1)
20     {
21         retval[7] |= 0x03;
22         retval[8] = 0x2a;
23     } else
24         retval[8] = 0xd5;
25
26     pn[0] = 0x03;
27     pn[1] = 0xF2;
28     pn[2] = 0xA3;
29     pn[3] = 0x3D;
30     pn[4] = 0xD6;
31     pn[5] = 0x9B;
32     pn[6] = 0x12;
33     pn[7] = 0x1C;
34     pn[8] = 0x10;
35
36     for(count = 4; count < 9; count++)
37         retval[count] ^= pn[count];
38
39     /* Codeword */
40     //g(d) = 0x585713DA9
41     data[0] = (retval[4] & 0x02) >> 1;
42     data[1] = (retval[4] & 0x01);
43     data[2] = (retval[5] & 0x80) >> 7;
44     data[3] = (retval[5] & 0x40) >> 6;
45     data[4] = (retval[5] & 0x20) >> 5;
46     data[5] = (retval[5] & 0x10) >> 4;
47     data[6] = (retval[5] & 0x08) >> 3;

```

```

47     data[7] = (retval[5] & 0x04) >> 2;
48     data[8] = (retval[5] & 0x02) >> 1;
49     data[9] = (retval[5] & 0x01);
50     data[10] = (retval[6] & 0x80) >> 7;
51     data[11] = (retval[6] & 0x40) >> 6;
52     data[12] = (retval[6] & 0x20) >> 5;
53     data[13] = (retval[6] & 0x10) >> 4;
54     data[14] = (retval[6] & 0x08) >> 3;
55     data[15] = (retval[6] & 0x04) >> 2;
56     data[16] = (retval[6] & 0x02) >> 1;
57     data[17] = (retval[6] & 0x01);
58     data[18] = (retval[7] & 0x80) >> 7;
59     data[19] = (retval[7] & 0x40) >> 6;
60     data[20] = (retval[7] & 0x20) >> 5;
61     data[21] = (retval[7] & 0x10) >> 4;
62     data[22] = (retval[7] & 0x08) >> 3;
63     data[23] = (retval[7] & 0x04) >> 2;
64     data[24] = (retval[7] & 0x02) >> 1;
65     data[25] = (retval[7] & 0x01);
66     data[26] = (retval[8] & 0x80) >> 7;
67     data[27] = (retval[8] & 0x40) >> 6;
68     data[28] = (retval[8] & 0x20) >> 5;
69     data[29] = (retval[8] & 0x10) >> 4;
70
71     cw = codeword(data, 64, 30);
72
73     retval[0] = cw[0] << 3 | cw[1] << 2 |
74                cw[2] << 1 | cw[3];
75
76     retval[1] = cw[4] << 7 | cw[5] << 6 |
77                cw[6] << 5 | cw[7] << 4 |
78                cw[8] << 3 | cw[9] << 2 |
79                cw[10] << 1 | cw[11];

```

```

80
81     retval[2] = cw[12] << 7 | cw[13] << 6 |
82                cw[14] << 5 | cw[15] << 4 |
83                cw[16] << 3 | cw[17] << 2 |
84                cw[18] << 1 | cw[19];
85
86     retval[3] = cw[20] << 7 | cw[21] << 6 |
87                cw[22] << 5 | cw[23] << 4 |
88                cw[24] << 3 | cw[25] << 2 |
89                cw[26] << 1 | cw[27];
90
91     retval[4] = cw[28] << 7 | cw[29] << 6 |
92                cw[30] << 5 | cw[31] << 4 |
93                cw[32] << 3 | cw[33] << 2 |
94                (retval[4] & 0x3);
95     free(cw);
96
97     for(count = 0; count < 9; count++)
98         retval[count] ^= pn[count];
99     free(pn);
100
101     /* Preamble */
102     if(retval[0] & 0x08)
103         retval[0] |= 0xa0;
104     else
105         retval[0] |= 0x50;
106
107     return retval;
108 }

```

E.1.3 checkac()

```

1  /* Create an AC and check it*/
2  bool bluetooth_UAP::check_ac(char *stream)

```

```

3 {
4     int count, aclength;
5     uint8_t *ac, *grdata;
6     aclength = 72;
7
8     /* Generate AC */
9     ac = acgen(dLAP);
10
11    /* Check AC */
12    /* Convert it to 1 LSB per byte */
13    grdata = (uint8_t *) malloc(aclength);
14
15    for(count = 0; count < 9; count++)
16        convert_to_grformat(ac[count], &grdata[count*8]);
17
18    for(count = 0; count < aclength; count++)
19    {
20        if(grdata[count] != stream[count])
21            return 0;
22    }
23    return 1;
24 }

```

E.1.4 sniffac()

```

1 /* Looks for an AC in the stream */
2 int bluetooth_UAP::sniff_ac()
3 {
4     int jump, count, counter, size;
5     char *stream = d_stream;
6     int jumps[16] = {3,2,1,3,3,0,2,3,3,2,0,3,3,1,2,3};
7     size = d_stream_length;
8     count = 0;
9

```

```
10     while(size > 72)
11     {
12         jump = jumps[stream[0] << 3 | stream[1] << 2 |
13             stream[2] << 1 | stream[3]];
14         if(0 == jump)
15         {
16             /* Found the start, now check the end... */
17             counter = stream[62] << 9 | stream[63] << 8 |
18                 stream[64] << 7 | stream[65] << 6 |
19                 stream[66] << 5 | stream[67] << 4 |
20                 stream[68] << 3 | stream[69] << 2 |
21                 stream[70] << 1 | stream[71];
22
23             if((0x0d5 == counter) || (0x32a == counter))
24             {
25                 if(check_ac(stream))
26                     return count;
27             }
28             jump = 1;
29         }
30         count += jump;
31         stream += jump;
32         size -= jump;
33     }
34     return -1;
35 }
```

E.2 Header

E.2.1 hecgen()

```
1  /* Pointer to start of header, UAP */
2  uint8_t hecgen(char *packet, int UAP)
3  {
4      char byte;
5      uint8_t reg, retval;
6      int count;
7
8      reg = UAP & 0xff;
9      byte = *packet++;
10
11     for(count = 0; count < 10; count++)
12     {
13         if(8==count)
14             byte = *packet;
15
16         reg = (reg << 1) | (((reg & 0x80) >> 7)
17             ^ (byte & 0x1));
18         byte >>= 1;
19
20         /* Bit 1*/
21         reg ^= ((reg & 0x01)<<1);
22         /* Bit 2*/
23         reg ^= ((reg & 0x01)<<2);
24         /* Bit 5*/
25         reg ^= ((reg & 0x01)<<5);
26         /* Bit 7*/
27         reg ^= ((reg & 0x01)<<7);
28     }
29     return reg;
```

```
30 }
```

E.2.2 UAP from HEC

```
1  /* Pointer to start of header, UAP */
2  int bluetooth_UAP::UAP_from_hec(uint8_t *packet)
3  {
4      char byte;
5      int count;
6      uint8_t hec;
7
8      hec = *(packet + 2);
9      byte = *(packet + 1);
10
11     for(count = 0; count < 10; count++)
12     {
13         if(2==count)
14             byte = *packet;
15
16         /* Bit 1*/
17         hec ^= ((hec & 0x01)<<1);
18         /* Bit 2*/
19         hec ^= ((hec & 0x01)<<2);
20         /* Bit 5*/
21         hec ^= ((hec & 0x01)<<5);
22         /* Bit 7*/
23         hec ^= ((hec & 0x01)<<7);
24
25         hec = (hec >> 1) | (((hec & 0x01)
26             ^ (byte & 0x01)) << 7);
27         byte >>= 1;
28     }
29     return hec;
30 }
```

E.2.3 Unwhiten header

```

1  void unwhiten_header(uint8_t *input, uint8_t *output, int clock)
2  {
3      int count, index;
4      /* index into data array */
5      uint8_t indicies[64] = {99, 85, 17, 50, 102, 58, 108, 45, 92,
6                             62, 32, 118, 88, 11, 80, 2, 37, 69, 55, 8, 20, 40,
7                             74, 114, 15, 106, 30, 78, 53, 72, 28, 26, 68, 7, 39,
8                             113, 105, 77, 71, 25, 84, 49, 57, 44, 61, 117, 10,
9                             1, 123, 124, 22, 125, 111, 23, 42, 126, 6, 112, 76,
10                            24, 48, 43, 116, 0};
11
12     /* whitening data */
13     uint8_t data[127] = {1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0,
14                          0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1,
15                          0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0,
16                          1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0,
17                          0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0,
18                          0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
19                          1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0,
20                          1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1};
21
22     index = indicies[clock & 0x3f];
23
24     for(count = 0; count < 18; count++)
25     {
26         output[count] = input[count] ^ data[index];
27         index += 1;
28         index %= 127;
29     }
30 }

```

E.3 Payload

E.3.1 crcgen()

```
1  uint16_t bluetooth_UAP::crcgen(char *packet, int length, int UAP)
2  {
3      char byte;
4      uint16_t reg, count, counter;
5
6      reg = UAP & 0xff;
7      for(count = 0; count < length; count++)
8      {
9          byte = *packet++;
10         for(counter = 0; counter < 8; counter++)
11         {
12             reg = (reg << 1) | (((reg & 0x8000)>>15)
13                 ^ ((byte & 0x80) >> 7));
14             byte <<= 1;
15
16             /* Bit 5*/
17             reg ^= ((reg & 0x0001)<<5);
18
19             /* Bit 12*/
20             reg ^= ((reg & 0x0001)<<12);
21         }
22     }
23     return reg;
24 }
```

E.4 Common functions

E.4.1 reverse

```
1  /* Reverse the bits in a byte */
2  uint8_t bluetooth_UAP::reverse(char byte)
3  {
4      return (byte & 0x80) >> 7 | (byte & 0x40) >> 5 |
5          (byte & 0x20) >> 3 | (byte & 0x10) >> 1 |
6          (byte & 0x08) << 1 | (byte & 0x04) << 3 |
7          (byte & 0x02) << 5 | (byte & 0x01) << 7;
8  }
```

E.4.2 convert to one LSB per byte format

```
1  void bluetooth_UAP::convert_to_grformat
2      (uint8_t input, uint8_t *output)
3  {
4      int count;
5      for(count = 0; count < 8; count++)
6      {
7          output[count] = (input & 0x80) >> 7;
8          input <<= 1;
9      }
10 }
```

Bibliography

- [1] <http://www.bluetooth.org>
- [2] Busy as a ZigBee, Jon Adams and Bob Heile
- [3] <http://standards.ieee.org/getieee802/download/802.15.4-2006.pdf>
- [4] A Preliminary Investigation of Worm Infections in a Bluetooth Environment by Jing Su, Kelvin K. W. Chan, Andrew G. Miklas, Kenneth Po, Ali Akhavan, Stefan Saroiu, Eyal de Lara, Ashvin Goel
- [5] M. Jakobsson and S. Wetzel. Security Weaknesses in Bluetooth. CT-RSA 2001: Proceedings of the 2001 Conference on Topics in Cryptology, pages 176191, 2001. LNCS 2020.
- [6] <https://programs.bluetooth.org/apps/faq/faq-details.aspx?fid=98>
- [7] Bluetooth Core Specification v1.2, Bluetooth Special Interest Group
- [8] http://www.ettus.com/downloads/transceiver_dbrds_v3b.pdf
- [9] <http://www.bluez.org>
- [10] <http://www.opencores.org/projects.cgi/web/bluetooth/overview>
- [11] email from Matt Ettus, designer of the USRP, <http://lists.hpsdr.org/pipermail/hpsdr-hpsdr.org/2006-November/002997.html>
- [12] United States Patent 6941110, Kloper, David S., Diener, Neil R., Cognio, Inc.

- [13] Yaniv Shaked, Avishai Wool (2005-06). "Cracking the Bluetooth PIN". School of Electrical Engineering Systems
<http://www.eng.tau.ac.il/~yash/shaked-wool-mobisys05/index.html>
- [14] http://trifinite.org/trifinite_stuff.html
- [15] <http://www.fte.com/getFTS4BT/>
- [16] standards.ieee.org/regauth/oui/oui.txt
- [17] <http://www.ettus.com/>
- [18] Busting The Bluetooth Myth Getting RAW Access, by Max Moser